目錄

面对对象编程语言：应用和解释

> 来源：mrmathematica/ooplai-zh

作者 Éric Tanter

译者 MrMathematica、lotuc

2017年版

本书以建设性和渐进的方式展示了面向对象编程语言的基本概念。它遵循Shriram Krishnamurthi的《编程语言：应用和解释》一书的方法（至少我打算这样做）。本文假定读者熟悉PLAI以下部分：一等函数，词法范围，递归和状态，以及宏。

致谢：我感谢PLEIAD实验室的成员和智利大学程序设计语言课程的学生，特别是Ricardo Honorato，他们发现了错误并提出了改进建议。

1 从函数到简单对象

对面向对象编程语言的探索从我们于PLAI（《编程语言：应用和解释》）中所学到的、以及对于什么是对象的直觉开始。

1.1 有状态函数与对象模式

对象的目的是，将状态（可能但不一定是可变的）连同依赖于该状态的行为一起封装在一致的整体中。这里的状态通常被称为字段(field)（或实例变量(instance variable)），而行为是以方法(method)的形式提供。调用方法通常被称为消息传递(message passing)：发送消息给对象，如果它理解了，就执行相关的方法。

在Scheme这样的高级编程语言中，我们看到过类似的东西：

```
(define add
  (λ (n)
    (λ (m)
      (+ m n))))

> (define add2 (add 2))
> (add2 5)
7
```

函数add2封装了隐藏状态（`n = 2`），其行为也依赖于该状态。所以从某种意义上说，闭包是一种对象，他的字段是（函数体中的）自由变量。那么其行为呢？好吧，闭包只有一个行为，通过函数调用触发（从消息传递的角度来看，apply(调用)是函数能理解的唯一消息）。

如果语言支持赋值（`set!`），那么我们就得到了有状态的函数，可以改变状态：

```
(define counter
  (let ([count 0])
    (λ ()
      (begin
        (set! count (add1 count))
        count))))
```

现在我们可以观察到count状态的变化：

```
> (counter)
1
> (counter)
2
```

现在，如果我们想要双向计数器呢？该函数必须能够在其状态上执行+1或者-1，取决于……好吧，参数！

```
(define counter
  (let ([count 0])
    (λ (cmd)
      (case cmd
        [(dec) (begin
                 (set! count (sub1 count))
                 count)]
        [(inc) (begin
                 (set! count (add1 count))
                 count)]))))
```

请注意counter如何使用cmd来区分要执行的操作。

```
> (counter 'inc)
1
> (counter 'dec)
0
```

这看起来很像有两个方法和一个实例变量的对象，不是吗？ 我们再来看一个例子，堆栈。

```
(define stack
  (let ([vals '()])
    (define (pop)
      (if (empty? vals)
          (error "cannot pop from an empty stack") ;无法从空栈中po
p
          (let ([val (car vals)])
            (set! vals (cdr vals))
            val)))

    (define (push val)
      (set! vals (cons val vals)))

    (define (peek)
      (if (empty? vals)
          (error "cannot peek from an empty stack") ;无法从空栈中p
eek
          (car vals)))

    (λ (cmd . args)
      (case cmd
        [(pop) (pop)]
        [(push) (push (car args))]
        [(peek) (peek)]
        [else (error "invalid command")])))) ;无效的命令
```

这里，我们没有直接在lambda中编写方法体，而是使用了内层的define。另外请注意，我们在lambda的参数中使用了点符号：这样函数就能够接收一个参数（cmd）以及零或多个额外参数（以链表形式在函数体中绑定到args）。

试试看：

```
> (stack 'push 1)
> (stack 'push 2)
> (stack 'pop)
2
> (stack 'peek)
1
> (stack 'pop)
1
> (stack 'pop)
cannot pop from an empty stack
```

这代码的模式已经很明显了，可以用来定义类似于对象的抽象。更明确地抽象此模式：

```
(define point
  (let ([x 0])
    (let ([methods (list (cons 'x? (λ () x))
                         (cons 'x! (λ (nx) (set! x nx))))])
      (λ (msg . args)
        (apply (cdr (assoc msg methods)) args)))))
```

请注意这里定义的λ，它以一种通用的方式将消息分发到正确的方法。我们首先把所有的方法都放在一个关联链表（即元素为pair的链表）中，将符号（也就是消息）关联到相应的方法。当调用point时，我们（用assoc）查找消息，得到相应的方法。然后调用它。

```
> (point 'x! 6)
> (point 'x?)
6
```

1.2 Scheme中的（第一种）简单对象系统

我们可以用宏在Scheme中嵌入一个遵循上面确定的模式的简单对象系统。

> 请注意，在本书中我们使用defmac来定义宏。defmac类似于 `define-syntax-rule` ，但是它还支持关键字参数，外加标识符捕获（通过 `#:keywords` 和 `#:captures` 可选参数）。

```
(defmac (OBJECT ([field fname init] ...)
                ([method mname args body] ...))
  #:keywords field method
  (let ([fname init] ...)
    (let ([methods (list (cons 'mname (λ args body)) ...)])
      (λ (msg . vals)
        (apply (cdr (assoc msg methods)) vals)))))
```

我们还可以定义箭头 `->` 符号表示发送消息给对象，例如 `(-> st push 3)` ：

```
(defmac (-> o m arg ...)
  (o 'm arg ...))
```

现在就可以使用这个对象系统来定义二维点对象了：

```
(define p2D
  (OBJECT
   ([field x 0]
    [field y 0])
   ([method x? () x]
    [method y? () y]
    [method x! (nx) (set! x nx)]
    [method y! (ny) (set! y ny)])))
```

这么使用：

```
> (-> p2D x! 15)
> (-> p2D y! 20)
> (-> p2D x?)
15
> (-> p2D y?)
20
```

1.3 构造对象

到目前为止，我们的对象都是作为独立样本被创建。如果我们想要多个点对象，每个可以有不同的初始坐标呢？

在函数式编程的语境中，我们知道如何正确地创建各种类似的函数：使用高阶函数，带上合适的参数，其作用是返回我们想要的特定实例。例如，从前面定义的add函数中，我们可以获得各种单参数加法函数：

```
> (define add4 (add 4))
> (define add5 (add 5))
> (add4 1)
5
> (add5 1)
6
```

因为我们的简单对象系统根植于Scheme，所以可以简单地使用高阶函数来定义对象工厂（object factory）：

> JavaScript，AmbientTalk

```
(define (make-point init-x init-y)
  (OBJECT
   ([field x init-x]
    [field y init-y])
   ([method x? () x]
    [method y? () y]
    [method x! (new-x) (set! x new-x)]
    [method y! (new-y) (set! y new-y)])))
```

`make-point` 函数的参数是初始坐标，返回新创建的、正确地初始化后的对象。

```
> (let ([p1 (make-point 5 5)]
        [p2 (make-point 10 10)])
    (-> p1 x! (-> p2 x?))
    (-> p1 x?))

10
```

1.4 动态分发

我们的简单对象系统就足以展示面向对象编程的基本特性：动态分发。请注意，在下面的代码中，node（节点）将sum消息发送给每个子节点，并不知道它们是leaf（叶节点）还是node：

```
(define (make-node l r)
 (OBJECT
  ([field left l]
   [field right r])
  ([method sum () (+ (-> left sum) (-> right sum))])))

(define (make-leaf v)
 (OBJECT
  ([field value v])
  ([method sum () value])))


> (let ([tree (make-node
               (make-node (make-leaf 3)
                          (make-node (make-leaf 10)
                                     (make-leaf 4)))
               (make-leaf 1))])
   (-> tree sum))

18
```

尽管看起来很简单，这个对象系统已经足以说明对象的基本抽象机制，以及它和抽象数据类型（abstract data type）的区别。参见第三章。

1.5 错误处理

让我们看看，如果发送消息给不知道如何处理它的对象会发生什么：

```
> (let ([l (make-leaf 2)])
    (-> l print))
cdr: contract violation
  expected: pair?
  given: #f
```

这个错误信息很糟糕——它将我们的实现策略暴露给程序员，而且没有提示问题在哪。

我们可以改变OBJECT语法抽象的定义，正确地处理未知消息：

```
(defmac (OBJECT ([field fname init] ...)
                ([method mname args body] ...))
  #:keywords field method
  (let ([fname init] ...)
    (let ([methods (list (cons 'mname (λ args body)) ...)])
      (λ (msg . vals)
        (let ([found (assoc msg methods)])
          (if found
              (apply (cdr found) vals)
              (error "message not understood:" msg))))))) ;未知的
消息
```

我们不再假设在对象的方法表中会有消息关联的方法，而是首先查找并将结果绑定到found；如果找不到方法，found将会是#f。在这种情况下，我们给出有意义的错误信息。

确实好多了：

```
> (let ([l (make-leaf 2)])
    (-> l print))
message not understood: print
```

本章，我们成功地在Scheme中嵌入了一个简单的对象系统，它显示了词法作用域的一等函数和对象之间的连接。但是，我们还远没有完成，目前的对象系统仍然不完整且非常原始。

2 寻找Self

在前一章中，我们构建了一个简单的对象系统。现在我们来考虑为点对象定义above方法，它读入的参数是另一个点，返回更高的（从y轴角度看）点：

```
(method above (other-point)
        (if (> (-> other-point y?) y)
            other-point
            self))
```

请注意，我们直观地使用self来表示当前正在执行的对象；在其他有些语言中，它被称为this。显然，我们对OOP的描述并没有告诉我们self是什么。

2.1 Self是什么?

回过头看看最初那个对象的定义（没有宏的那个）。对象是函数；所以我们想要的是在这个函数范围内能够引用自己。该怎么做呢？研究递归的时候我们已经知道答案了！只需使用递归绑定（letrec）给函数—对象命名，然后就可以在方法定义中使用了：

```
(define point
  (letrec ([self
            (let ([x 0])
              (let ([methods (list (cons 'x?  (λ () x))
                                   (cons 'x! (λ (nx)
                                               (set! x nx)
                                               self)))])
                (λ (msg . args)
                  (apply (cdr (assoc msg methods)) args))))])
    self))
```

请注意，letrec的主体就返回self，它绑定到我们定义的递归子程序。

```
> ((point 'x! 10) 'x?)
10
```

> 在Smalltalk语言中，方法默认返回self。

请注意，赋值方法 `x!` 返回self，这使得我们可以链式传递消息。

2.2 用宏实现Self

在我们的OBJECT宏中使用上述模式：

```
(defmac (OBJECT ([field fname init] ...)
                ([method mname args body] ...))
  #:keywords field method
  (letrec ([self
            (let ([fname init] ...)
              (let ([methods (list (cons 'mname (λ args body)) .
..)])
                (λ (msg . vals)
                  (apply (cdr (assoc msg methods)) vals))))])
    self))

(defmac (-> o m arg ...)
  (o 'm arg ...))
```

用一些点对象试试：

```
(define (make-point init-x)
  (OBJECT
   ([field x init-x])
   ([method x? () x]
    [method x! (nx) (set! x nx)]
    [method greater (other-point)
            (if (> (-> other-point x?) x)
                other-point
                self)])))

> (let ([p1 (make-point 5)]
        [p2 (make-point 2)])
    (-> p1 greater p2))
self: undefined;
 cannot reference undefined identifier
```

什么？？我们明明用letrec定义了self，为什么报错说它没有定义呢？原因是——卫生！要知道Scheme的syntax-rules是卫生的，因此，它会透明地重命名宏引入的所有标识符，以确保在宏展开后他们不会意外绑定或者被绑定。使用DrRacket的宏步进器（macro stepper）可以很清楚地观察到这一点。你会看到，greater方法中的self标识符与letrec表达式中的同名标识符的颜色不同。

幸运的是，defmac支持一种方法，指定宏本身引入的标识符也可以被用户代码使用。这里我们唯一需要做的是指定self就是这样的标识符：

```
(defmac (OBJECT ([field fname init] ...)
                ([method mname args body] ...))
  #:keywords field method
  #:captures self
  (letrec ([self
            (let ([fname init] ...)
              (let ([methods (list (cons 'mname (λ args body)) .
..)])
                (λ (msg . vals)
                  (apply (cdr (assoc msg methods)) vals))))])
    self))
```

2.3 用到Self的点对象

现在我们可以定义种种方法，或返回self，或在方法体中使用self：

```
(define (make-point init-x init-y)
 (OBJECT
  ([field x init-x]
   [field y init-y])
  ([method x? () x]
   [method y? () y]
   [method x! (new-x) (set! x new-x)]
   [method y! (new-y) (set! y new-y)]
   [method above (other-point)
           (if (> (-> other-point y?) y)
               other-point
               self)]

   [method move (dx dy)
           (begin (-> self x! (+ dx (-> self x?)))
                  (-> self y! (+ dy (-> self y?)))
                  self)])))

(define p1 (make-point 5 5))
(define p2 (make-point 2 2))

> (-> (-> p1 above p2) x?)
5
> (-> (-> p1 move 1 1) x?)
6
```

2.4 互相递归的方法

上一节已经表明，方法可以通过向self发送消息来使用其他方法。这个例子展示相互递归的方法。

> 请在Java中尝试相同的定义，然后比较“大”数字的结果。是啊，我们的简单对象系统确实从尾调用优化中受益了！

```
(define odd-even
  (OBJECT ()
   ([method even (n)
            (case n
              [(0) #t]
              [(1) #f]
              [else (-> self odd (- n 1))])]
    [method odd (n)
            (case n
              [(0) #f]
              [(1) #t]
              [else (-> self even (- n 1))])])))


> (-> odd-even odd 15)
#t
> (-> odd-even odd 14)
#f
> (-> odd-even even 14)
#t
```

我们现在的对象系统支持self，包括返回self、发送消息给self。请注意，方法中使用的self是在对象创建时被绑定的：在方法被定义时，它们捕获对self的绑定，此后该绑定就被固定了。我们将在下面的章节中看到，如果想要支持委托，或者想要支持类，这就行不通了。

2.5 嵌套的对象

对象和方法最终被编译成Scheme中的lambda，因此我们的对象继承了一些有趣的属性。首先，正如我们所看到的，它们是一等公民（不然这一切还有意思吗？）。另外，正如我们刚刚看到的，尾位置的方法调用被视为尾调用，因此空间没有浪费。接下来讨论另一个好处：我们可以使用高阶的编程模式，比如产生对象的对象（通常称为工厂）。换一种说法，运用合适的词法范围，我们可以定义嵌套的对象。

考虑如下的例子：

```
(define factory
  (OBJECT
   ([field factor 1]
    [field price 10])
   ([method factor! (nf) (set! factor nf)]
    [method price! (np) (set! price np)]
    [method make ()
            (OBJECT ([field price price])
                    ([method val () (* price factor)]))])))

> (define o1 (-> factory make))
> (-> o1 val)
10
> (-> factory factor! 2)
> (-> o1 val)
20
> (-> factory price! 20)
> (-> o1 val)
20
> (define o2 (-> factory make))
> (-> o2 val)
40
```

> 在Java中你能这么做吗？

请验证这些返回。

3 对象的好处和局限性

在编程语言的课程中，我们这样编程：定义数据类型及其变体，在此之上定义操作这些结构体的各种“服务”，所谓服务也即对这些数据结构的各种变种分情况进行处理的子程序。这种编程风格有时被称为“过程式”或“函数设计”的（注意这里的“函数”并不是指“无副作用的”！）。

在《程序语言：应用和解释》中，我们用 `define-type` 来定义数据类型及其变体，用 `type-case` 实现对变种按情况处理的子程序。这种编程方法在其他语言中也很常见：C（联合体）、Pascal（变体类型）、ML和Haskell的代数数据类型、纯Scheme的带标记数据。

那么，面向对象编程究竟给我们提供了什么呢？它的缺点是什么呢？事实证明，使用面向对象的语言并不意味着程序就是“面向对象”的。许多Java程序就不是，或者至少是牺牲了对象的某些基本好处的。

> 本章基于William R. Cook 2009年的《On Understanding Data Abstraction, Revisited》（再谈对数据抽象的理解）一文。

本独立章节的目的是，暂时从逐步构建OOP的步骤中抽身，转而对比面向对象和过程式编程，从而明确每种方法各自的优缺点。有趣的是，我们迄今为止构建的简单对象系统完全足够研究对象的基本好处和局限性了——委托、类、继承等都是有趣的特性，但对于对象来说都不是本质的。

3.1 抽象数据类型

我们先来讨论抽象数据类型（ADT）。ADT是隐藏其表示、只提供对值的操作的数据类型。

例如，整数的集合ADT可以定义如下：

```
adt Set 是
  empty : Set
  insert : Set x Int -> Set
  isEmpty? : Set -> Bool
  contains? : Set x Int -> Bool
```

这种整数集ADT有许多可能的表示。例如，可以使用Scheme的表来实现它：

```
(define empty '())

(define (insert set val)
  (if (not (contains? set val))
      (cons val set)
      set))

(define (isEmpty? set) (null? set))

(define (contains? set val)
  (if (null? set) #f
      (if (eq? (car set) val)
          #t
          (contains? (cdr set) val))))
```

客户程序可以使用ADT值，而无需知道底层的表示法：

```
> (define x empty)
> (define y (insert x 3))
> (define z (insert y 5))
> (contains? z 2)
#f
> (contains? z 5)
#t
```

我们也可以用另一种表示方式来实现ADT集合，比如使用PLAI的define-type机制来创建一个变体类型，将集合编码为链表。

```
(define-type Set
  [mtSet]
  [aSet (val number?) (next Set?)])

(define empty (mtSet))

(define (insert set val)
  (if (not (contains? set val))
      (aSet val set)
      set))

(define (isEmpty? set) (equal? set empty))

(define (contains? set val)
  (type-case Set set
    [mtSet () #f]
    [aSet (v next)
          (if (eq? v val)
              #t
              (contains? next val))]))
```

前面的示例客户程序运行照旧，即使现在底层表示换掉了：

```
> (define x empty)
> (define y (insert x 3))
> (define z (insert y 5))
> (contains? z 2)
#f
> (contains? z 5)
#t
```

3.2 用子程序表示

我们也可以把集合看作是由它的特征函数定义：该函数读入一个数字，告诉我们这个数字是否是集合的一部分。在这种情况下，集合就是简单的 `Int -> Bool` 函数。（PLAI一书中，第十二章中在研究环境的子程序表示时有提到。）

空集的特征函数是什么？总是返回假的函数。插入一个新元素所获得的集合呢？

```
(define empty (λ (n) #f))

(define (insert set val)
          (λ (n)
            (or (eq? n val)
                (contains? set n))))

(define (contains? set val)
  (set val))
```

由于集合由其特征函数表示， `contains?` 只需将该函数应用于该元素。请注意，客户程序还是完全不受干扰：

```
> (define x empty)
> (define y (insert x 3))
> (define z (insert y 5))
> (contains? z 2)
#f
> (contains? z 5)
#t
```

集合的子程序表示给我们带了了什么？灵活性！例如，我们可以定义所有偶数的集合：

```
(define even
  (λ (n) (even? n)))
```

我们前面考虑的任何ADT表示，都不能完整地表示这个集合。（为什么？）我们甚至可以定义非确定的集合：

```
(define random
  (λ (n) (> (random) 0.5)))
```

使用子程序表示，我们可以更自由地定义集合，此外它们同样可以与已有的集合操作交互！

```
> (define a (insert even 3))
> (define b (insert a 5))
> (contains? b 12)
#t
> (contains? b 5)
#t
```

相反，在上面我们看到的ADT表示中，不同的表示法之间不能互操作。列表实现集合的值不能被结构体实现的操作使用，反之亦然。ADT从表示中抽象出来，但一次只允许一种表示。

3.3 对象

从本质上讲，函数实现的集合就是对象！请注意对象并未抽象出类型：函数实现的集合的类型非常具体：它是 `Int -> Bool` 的函数。当然，正如我们在前面的章节中看到的，对象是函数的泛化，它可以有多个方法。

3.3.1 对象的接口

我们可以定义对象接口（interface）的概念，也就是某个对象所有方法的型签（类型签名，signature）：

```
interface Set 是
  contains? : Int -> Bool
  isEmpty? : Bool
```

使用我们的简单对象系统实现集合对象：

```
(define empty
  (OBJECT ()
          ([method contains? (n) #f]
           [method isEmpty? () #t])))

(define (insert s val)
  (OBJECT ()
          ([method contains? (n)
                   (or (eq? val n)
                       (-> s contains? n))]
           [method isEmpty? () #f])))
```

请注意，empty是个对象，insert是返回对象的工厂函数。集合对象实现了Set接口。 `empty` 对象不包含任何值，它的 `isEmpty？` 返回 `#t` 。 `insert` 返回一个新对象，它的 `contains?` 方法类似于前文中集合的特征函数，而 `isEmpty?` 返回 `#f` 。

客户程序中，构造集合部分不用变，与集合对象交互部分就必须用消息发送了：

```
> (define x empty)
> (define y (insert x 3))
> (define z (insert y 5))
> (-> z contains? 2)
#f
> (-> z contains? 5)
#t
```

请注意，对象接口本质上就是高阶类型：方法是函数，所以传递对象就是传递函数组。这是高阶函数式编程的推广。面向对象的程序本质上是高阶的。

3.3.2 面向对象编程的原则

原则：对象只能通过其他对象的公共接口来访问它们

一旦创建了对象，比如上面的z（所绑定的），对它唯一能做的就是通过发送消息进行交互。不能“打开对象”。对象的任何属性都不可见，可见的只有它的接口。换一种说法：

原则：对象只对自己有详细的了解

这与ADT值有本质区别：在 `type-case` 的处理中（回忆一下ADT实现中用 `define-type` 实现的 `contains?` ），我们打开值，从而直接访问其属性。ADT提供封装，但为ADT的客户提供；不为其实现提供。对象在这方面更进一步。即使是对象的方法，其实现也不能访问除自身以外对象的属性。

由此我们可以得出另一个基本原则：

原则：对象就是所有对其可能进行的观测的集合，这些观测通过对象接口定义

这是一条强原则，它表明，如果两个对象在对于特定实验（即一组观测）表现相同，那么它们应该是不可区分的。这意味着使用等值判定操作（如指针相等）违反了OOP的这个原则。使用Java中的 `==` ，我们可以区分即使是行为一致的两个对象。

3.3.3 可扩展性

上述原则可以被认为是OOP的本质特征。正如Cook所说：“任何允许区分多个抽象表示的编程模型都不是面向对象的”。

组件对象模型（COM）是实践中最纯粹的OO编程模型之一。COM遵守上述所有的原则：没有内置的相等性，没有办法确定某个对象是否是某个类的实例。因此COM程序是高度可扩展的。

请注意，对象的可扩展性实际上完全独立于继承！（我们的语言甚至还没有类。）它来自对接口的使用。

3.3.4 那Java呢？

Java不是一种纯粹的面向对象的语言，并不是因为它有原始类型（primitive type，也有称作内置类型、基础类型或者基本类型），而是因为它支持的许多操作违反了我们上面描述的原则。Java内置支持相等 `==` 、 `instanceof` 、转换为类类型，这使得两个对象即使行为一致，也可以被区分。在Java中，可以声明一个方法，根据类来接受对象，而不是根据它们的接口（在Java中，类名也是类型）。当然还有就是，Java允许对象访问其他对象的内部（公有字段当然可以，但即使私有字段同一类的对象也可以访问！）。

这意味着Java也支持ADT风格的编程。这没有什么不对的！但重要的是了解这所涉及的设计上的取舍，然后做出明智的选择。例如，在JDK中，某些类在表面上尊重OO原则（允许可扩展性），但其实现使用ADT技术（不可扩展，但更高效）。如果你有兴趣，参见 `List` 接口和 `LinkedList` 实现。

在Java中，“纯OO”编程基本上就是不使用类名称作为类型（即只在 `new` 之后使用类名），并且从不使用内置的相等（ `==` ）。

3.4 可扩展性问题

面向对象程序设计通常被认为是软件可扩展性方面的灵丹妙药。但是，“可扩展”究竟意味着什么呢？

可扩展性问题说的是如何定义数据类型（结构＋操作），使之能够支持两种形式的扩展：添加新的表示变体，或添加新的操作。

> 这里，ADT的意思遵从Cook的用法。然而我们需要澄清，这里对扩展性问题的讨论实际上将对象与变体类型(variant type)（即代数数据类型(algebraic data types)）进行对比。我们关心的是可扩展的实现。这里不关心界面的抽象。

事实表明，ADT和对象分别都能很好地支持可扩展性的一个维度，但是在另一维度就不行了。让我们用一个众所周知的例子来研究此问题：简单表达式的解释器。

3.4.1 ADT

先来考虑ADT的做法。表达式的数据类型有三种变体：

```
(define-type Expr
   [num  (n number?)]
   [bool (b boolean?)]
   [add (l Expr?) (r Expr?)])
```

接下来定义解释器，这是一个函数，用type-case处理抽象语法树：

```
(define (interp expr)
   (type-case Expr expr
      [num (n) n]
      [bool (b) b]
      [add (l r) (+ (interp l) (interp r))]))
```

这是一道很好的PLAI练习题。举个例子：

```
> (define prog (add (num 1)
                    (add (num 2) (num 3))))
> (interp prog)
6
```

扩展：新的操作

先来考虑给表达式添加一个新操作。除了对表达式进行解释，我们还想做类型检查，也就是确定它将算得的值的类型（在这里，是 `number` 或 `boolean` ）。这很简单，但是能检测到解释过程中出现的失败的情况，比如对两个不是数字的东西进行相加操作：

```
(define (typeof expr)
  (type-case Expr expr
    [num (n) 'number]
    [bool (b) 'boolean]
    [add (l r) (if (and (equal? 'number (typeof l))
                        (equal? 'number (typeof r)))
                   'number
                   (error "类型错误：并非数"))]))
```

求一下之前那个程序的类型：

```
> (typeof prog)
'number
```

我们的类型检查器会拒绝不合理的程序：

```
> (typeof (add (num 1) (bool #f)))
类型错误：并非数
```

反思一下这个扩展案例，我们看到一切都很顺利。想要新的操作，我们只需要定义新的函数。这种扩展是模块化的，因为只需要在一个地方新加定义。

扩展：新的数据

接下来考虑另一个维度的可扩展性：添加新的数据变体。假设我们扩展这里的简单语言，增加新的表达式：`ifc`。扩展后数据类型的定义是：

```
(define-type Expr
  [num  (n number?)]
  [bool (b boolean?)]
  [add (l Expr?) (r Expr?)]
  [ifc (c Expr?) (t Expr?) (f Expr?)])
```

修改 `Expr` 的定义加上这个新变体破坏了所有现有的函数定义！`interp` 和 `typeof` 都不再成立，因为它们用 `type-case` 对表达式“按类型处理”，但是并没有处理 `ifc` 的情况。我们需要修改它们，加上对 `ifc` 的处理：

```
(define (interp expr)
  (type-case Expr expr
    [num (n) n]
    [bool (b) b]
    [add (l r) (+ (interp l) (interp r))]
    [ifc (c t f)
         (if (interp c)
             (interp t)
             (interp f))]))

(define (typeof expr)
  (type-case Expr expr
    [num (n) 'number]
    [bool (b) 'boolean]
    [add (l r) (if (and (equal? 'number (typeof l))
                        (equal? 'number (typeof r)))
                   'number
                   (error "类型错误：并非数"))]
    [ifc (c t f)
         (if (equal? 'boolean (typeof c))
             (let ((type-t (typeof t))
                   (type-f (typeof f)))
               (if (equal? type-t type-f)
                   type-t
                   (error "类型错误：两个分支的类型不同")))
             (error "类型错误：并非布尔值"))]))
```

程序是正确的：

```
> (define prog (ifc (bool false)
                    (add (num 1)
                         (add (num 2) (num 3)))
                    (num 5)))
> (interp prog)
5
```

这种情况下的可扩展性就不怎么样了。我们必须修改数据类型的定义，然后修改所有的函数。

总而言之，使用ADT，添加新的操作（如 `typeof` ）是模块化的所以很容易，但添加新的数据类型（例如 `ifc` ）则不是模块化的所以非常麻烦。

3.4.2 OOP

对象在这些场景下表现如何？

我们从面向对象版本的解释器开始：

```
(define (bool b)
  (OBJECT () ([method interp () b])))

(define (num n)
  (OBJECT () ([method interp () n])))

(define (add l r)
  (OBJECT () ([method interp () (+ (-> l interp)
                                   (-> r interp))])))
```

请注意，遵循面向对象的设计原则，每个表达式对象都知道如何解释自己。程序中不存在某个中央解释器能处理所有的表达式。解释程序是通过给该程序发送 `interp` 消息来完成：

```
> (define prog (add (num 1)
                    (add (num 2) (num 3))))
> (-> prog interp)
6
```

扩展：新的数据

要添加新的数据，比如条件对象ifc，可以简单地定义新的对象工厂，其中包含该新对象处理interp消息的定义：

```
(define (ifc c t f)
  (OBJECT () ([method interp ()
                      (if (-> c interp)
                          (-> t interp)
                          (-> f interp))])))
```

现在可以解释包含条件的程序了：

```
> (-> (ifc (bool #f)
           (num 1)
           (add (num 1) (num 3))) interp)
4
```

这表明，与ADT相反，使用OOP添加新类型的数据是直接的、模块化的：只需创建新对象即可。对比ADT，这是明显的优势。

扩展：新的操作

但在得出结论，认为OOP是软件可扩展性的灵丹妙药之前，我们必须考虑另一种扩展场景：添加操作。假设我们和以前一样，需要检查程序的类型。这意味着表达式对象现在还需要理解“typeof”消息。要做到这一点，我们就必须修改所有的对象定义：

```
(define (bool b)
  (OBJECT () ([method interp () b]
              [method typeof () 'boolean])))

(define (num n)
  (OBJECT () ([method interp () n]
              [method typeof () 'number])))

(define (add l r)
  (OBJECT () ([method interp () (+ (-> l interp)
                                   (-> r interp))]
              [method typeof ()
                      (if (and (equal? 'number (-> l typeof))
                               (equal? 'number (-> r typeof)))
                          'number
                          (error "类型错误：并非数"))])))

(define (ifc c t f)
  (OBJECT () ([method interp ()
                      (if (-> c interp)
                          (-> t interp)
                          (-> f interp))]
              [method typeof ()
                      (if (equal? 'boolean (-> c typeof))
                          (let ((type-t (-> t typeof))
                                (type-f (-> f typeof)))
                            (if (equal? type-t type-f)
                                type-t
                                (error "类型错误：两个分支的类型不同"
)))
                          (error "类型错误：并非布尔值"))])))
```

程序是正确的：

```
> (-> (ifc (bool #f) (num 1) (num 3)) typeof)
'number
> (-> (ifc (num 1) (bool #f) (num 3)) typeof)
类型错误：并非布尔值
```

这个可扩展性场景下，我们被迫修改所有的代码才能添加新方法。

总而言之，对对象来说，添加新的数据类型（例如ifc）模块化所以容易，但添加新的操作（例如typeof）不模块化所以麻烦。

请注意，这就是ADT的对偶情况！

3.5 不同形式的数据抽象

> Cook的论文更深入地讨论了此类数据抽象之间的比较，不可不看！

ADT和对象是不同形式的数据抽象，各有优劣。

ADT的表示类型是私有的，无法篡改或扩展。这对推理（分析）和优化来说是好的。但它（同时）只允许一种表示。

对象拥有行为接口，因此可以随时定义新的实现。这对灵活性和可扩展性来说是好的。但这使得分析代码变得困难，并且使某些优化成为不可能。

这两种抽象形式也支持不同形式的模块化扩展。在ADT上可以模块化地添加新操作，但是支持新的数据变体就很麻烦。面向对象的系统可以模块化地添加新的表示法，但添加新的操作意味着大量的修改。

有一些方法可以绕开此折衷。比如说，在对象的接口中可以公开某些实现细节。这会牺牲一些可扩展性，但恢复某些优化的可能性。所以，这里根本的问题是设计上的问题：我们究竟需要什么？

现在你可以明白，为什么许多语言（同时）支持这两种数据抽象。

4 转发和委托

如果一个对象不知道如何处理某条消息，总是可以通过发送消息的方式将其转发给另一个对象。在我们的简单对象系统中，可以这么做：

```
(define seller
 (OBJECT ()
  ([method price (prod)
           (* (case prod
                ((1) (-> self price1))
                ((2) (-> self price2)))
              (-> self unit))]
   [method price1 () 100]
   [method price2 () 200]
   [method unit () 1])))

(define broker
 (OBJECT
  ([field provider seller])
  ([method price (prod) (-> provider price prod)])))

> (-> broker price 2)
200
```

对象 `broker` (中间商)不知道如何计算产品(`prod` ，product)的价格(price)，但它可以声称自己能提供价格信息，而其做法就是实现一个方法处理 `price` 消息，然后是简单地将消息转发给 `seller` (卖方)，由 `seller` 实现所需的行为。请注意， `broker` 在其 `provider` (供应商)字段中保有对seller的引用。这是典型的对象组合的例子，通过消息转发实现。

现在我们可以看到这种方法的问题了：消息的转发必须显式给出，对于每种我们预计可能发送给 `broker` 的消息，都必须定义一个负责转发到 `seller` 的方法。例如：

```
> (-> broker unit)
message not understood: unit
```

4.1 消息转发

我们可以做得更好，让每个对象都有一个特殊的“伙伴”对象，任何不理解的消息都自动转发给它。可以定义新的语法抽象 `OBJECT-FWD` 用于构造这样的对象：

```
(defmac (OBJECT-FWD target
                    ([field fname init] ...)
                    ([method mname args body] ...))
  #:keywords field method
  #:captures self
  (letrec ([self
            (let ([fname init] ...)
              (let ([methods (list (cons 'mname (λ args body)) .
..)])
                (λ (msg . vals)
                  (let ([found (assoc msg methods)])
                    (if found
                        (apply (cdr found) vals)
                        (apply target msg vals)))))))])
    self))
```

请注意这里语法的扩展，指定了 `target` 对象；只要某条消息在对象的方法中找不到，调度过程就会使用 `target` 对象。当然，如果所有对象都将未知消息转发给其他对象，那么传递链中必须有个最后的对象，该对象在收到消息时可以简单报错：

```
(define root
  (λ (msg . args)
    (error "not understood" msg)))
```

于是 `broker` 可以这样定义：

```
(define broker
  (OBJECT-FWD seller () ()))
```

这就是说，`broker` 是个空对象（不含字段，不含方法），只是将所有发送给它的消息转发给 `seller` ：

```
> (-> broker price 2)
200
> (-> broker unit)
1
```

这种对象通常被称为代理（proxy）。

4.2 委托

假设我们想用 `broker` 来改善 `seller` 的行为；比方说，我们希望通过改变价格计算中使用的单位，来使每个产品的价格加倍。这很简单：我们只需要在 `broker` 中定义方法 `unit` (单位)：

```
(define broker
  (OBJECT-FWD seller ()
   ([method unit () 2])))
```

有了这个定义，我们应该确保向 `broker` 询问某个产品的价格是向 `seller` 询问同样产品价格的两倍：

```
> (-> broker price 1)
100
```

嗯......这样不行！看来，一旦我们把 `price` 消息转发给 `seller` ，控制权将不再能流回 `broker` ；这里也即， `seller` 发给 `self` 的 `unit` 消息不会被 `broker` 收到。

让我们考虑一下这是为什么。在 `seller` 中 `self` 绑定到哪个对象？ `seller` ！请记住，我们之前说过（参见寻找Self），在我们的方法中， `self` 是静态绑定的：当对象被创建时， `self` 指向正被定义的对象/闭包，并且将始终绑定该值。这是因为 `letrec` 和 `let` 一样，遵从词法作用域。

我们正在寻找的则是另一种语义，称为委托（delegation）。委托要求对象中的 `self` 动态绑定：它应该始终指向最初接收消息的对象。在我们的例子中，这将确保当 `seller` 向 `self` 发送 `unit` 消息时， `self` 指向 `broker` ，这样 `broker` 中新定义的 `unit` 将会生效。在这种情况下，我们说 `seller` 是 `broker` 的父对象（parent）， `broker` 委托父对象处理消息。

怎样绑定标识符，能使其指向使用位置的值，而不是定义位置？在语言不提供动态作用域绑定指令的情况下，唯一可以实现这一点的方法是将该值作为参数传递。所以，必须给方法增加参数，新参数指向实际的接收方(receiver)。因此，不再从静态作用域中捕获 `self` 标识符，我们添加 `self` 参数。

具体说来，这意味着 `seller` 中这个方法：

```
(λ (prod) .... (-> self unit) ....)
```

必须改为：

> 有没有想过为什么Python中的方法必须显式地接受self作为第一个参数？

```
(λ (self)
  (λ (prod)....(-> self unit)....))
```

这个新参数有效地允许我们在查找得到方法后传递当前的接收方。

现在让我们定义新的语法形式 `OBJECT-DEL` ，来支持对象之间的委托（delegation）语义：

```
(defmac (OBJECT-DEL parent
                    ([field fname init] ...)
                    ([method mname args body] ...))
  #:keywords field method
  #:captures self
  (let ([fname init] ...)
    (let ([methods
           (list (cons 'mname
                       (λ (self) (λ args body))) ...)])
      (λ (current)
        (λ (msg . vals)
          (let ([found (assoc msg methods)])
            (if found
                (apply ((cdr found) current) vals)
                (apply (parent current) msg vals))))))))
```

有几地方改动了：首先，`target` 更名为 `parent` ，以明确我们定义的是委托语义。其次，如上所述，所有的方法现在都是带上了 `self` 参数。请注意，我们完全摆脱了 `letrec` ！这是因为 `letrec` 本来的用途就是允许对象引用 `self` ，同时遵循词法作用域。我们已经看到，对于委托来说，我们并不想要词法作用域。

这意味着，当我们在方法字典中找到某个方法时，必须首先将实际的接收方作为参数传给它。我们如何获得接收方？唯一的可能就是，给对象也加上参数，新参数是调用其方法时必须使用的当前接收方。也就是说，对象构造器返回的值不再是“ `λ (msg . vals) ....` ”，而是“ `λ (rcvr) ....` ”。“当前接收方”是我们的对象的参数。同样，如果某个消息不能被给定的对象所理解，那么它必须把当前接收者一起发送给它的父对象。

这样我们还有最后一个问题要解决：如何向对象发送消息？回忆一下， `->` 的定义是：

```
(defmac (-> o m arg ...)
  (o 'm arg ...))
```

但是现在我们不能简单地把 `o` 当做函数来调用，传给它一个符号（消息）和可变数量的参数。现在，对象是形式为 `(λ (rcvr) (λ (msg . args) ....))` 的函数。所以在传递消息和参数之前，我们必须指定哪个对象是当前的接收方。好吧，这很容易，因为在我们发送消息的时候，当前的接收方应该是……接受消息的对象！

> 为什么这里需要let绑定？

```
(defmac (-> o m arg ...)
  (let ([obj o])
    ((obj obj) 'm arg ...)))
```

来看委托——也就是`self`的延迟绑定——的效果：

```
(define seller
 (OBJECT-DEL root ()
  ([method price (prod)
           (* (case prod
                [(1) (-> self price1)]
                [(2) (-> self price2)])
              (-> self unit))]
   [method price1 () 100]
   [method price2 () 200]
   [method unit () 1])))
(define broker
 (OBJECT-DEL seller ()
  ([method unit () 2])))

> (-> seller price 1)
100
> (-> broker price 1)
200
```

4.3 用原型编程

具有类似我们在本章中介绍的委托机制的基于对象的语言被称为基于原型的语言（prototype），例如Self，JavaScript和AmbientTalk等等。这些语言擅长什么？如何使用原型编程？

4.3.1 单例和特殊对象

由于对象可以无中生有地创建（即，用类似于`OBJECT-DEL`的对象字面表达式创建），所以自然地可以创建只包含一个实例的类型的对象实例。与基于类的语言需要一个特定的设计模式（称为单例(Singleton)）相反，基于对象的语言非常适合这种情况，也适合创建“特殊”对象（下面会详细介绍）。

我们先来考虑布尔值的面向对象表示和简单的`if-then-else`控制结构。有多少种布尔值？只有两个：真和假。所以我们可以创建两个独立的对象，`true`和`false`来表示它们。在像Self和Smalltalk这样的纯面向对象的语言中，像`if-then-else`，`while`等这样的控制结构在语言中不是基本指令。相反，它们被定义为某些对象的方法。我们来考虑`if-then-else`的情况。我们可以给一个布尔值传两个thunk（译注，无参数的lambda，即`(lambda () ...)`），一个真thunk和一个假thunk；如果布尔值是true，它会调用真thunk；如果它是false，它会调用假thunk。

```
(define true
  (OBJECT-DEL root ()
    ([method ifTrueFalse (t f) (t)])))

(define false
  (OBJECT-DEL root ()
    ([method ifTrueFalse (t f) (f)])))
```

怎么能使用这些对象？举个例子：

```
(define light
 (OBJECT-DEL root
   ([field on false])
   ([method turn-on () (set! on true)]
    [method turn-off () (set! on false)]
    [method on? () on])))


> (-> (-> light on?) ifTrueFalse (λ () "灯开了")
                                 (λ () "灯关了"))
"灯关了"
> (-> light turn-on)
> (-> (-> light on?) ifTrueFalse (λ () "灯开了")
                                 (λ () "灯关了"))
"灯开了"
```

对象 `true` 和 `false` 是布尔值的唯二表示。任何依赖某个表达式为真或假的条件机制都可以类似地定义为这两个对象的方法。这就是动态分发！

Smalltalk中的布尔值和控制结构就是这么定义的，不过，由于Smalltalk是基于类的语言，它们的定义更加复杂些。用你最喜欢的基于类的语言来试试看。

我们再来看一个基于对象语言的实用例子：特殊（exceptional）对象。先来回顾一下普通点对象的定义，一般是调用工厂函数 `make-point` 创建的：

```
(define (make-point x-init y-init)
  (OBJECT-DEL root
    ([field x x-init]
     [field y y-init])
    ([method x? () x]
     [method y? () y])))
```

假设我们要引入一个特殊的点对象，它的特殊性在于坐标是随机的，每次访问都会改变。我们可以简单地定义 `random-point` 为一个独立的对象，其 `x?` 和 `y?` 方法执行计算而不是访问存储的状态：

```
(define random-point
  (OBJECT-DEL root ()
    ([method x? () (* 10 (random))]
     [method y? () (-> self x?)])))
```

请注意，`random-point`没有声明任何字段。当然，因为在OOP中我们依赖的是对象的接口，两种表示可以共存。

4.3.2 通过委托共享

上面讨论的例子突出了基于对象的语言的优点。现在让我们看看实际使用中的委托。首先，委托可以用来分解对象之间的共享行为。考虑这种情况：

```
(define (make-point x-init y-init)
  (OBJECT-DEL root
    ([field x x-init]
     [field y y-init])
    ([method x? () x]
     [method y? () y]
     [method above (p2)
             (if (> (-> p2 y?) (-> self y?))
                 p2
                 self)]
     [method add (p2)
             (make-point (+ (-> self x?)
                            (-> p2 x?))
                         (+ (-> self y?)
                            (-> p2 y?)))])))
```

创建的所有点对象都具有相同的方法，因此这些行为可以移至公共的父对象（通常称为原型）中，以实现共享。所有的行为都应该移到原型中吗？如果我们想要允许点的不同表示，比如前面的随机点（它根本不含任何字段！），就不该这么做。

因此，我们可以定义`point`原型，它提取了`above`和`add`方法，它们的实现对所有点都是一样的：

```
(define point
  (OBJECT-DEL root ()
    ([method above (p2)
             (if (> (-> p2 y?) (-> self y?))
                 p2
                 self)]
     [method add (p2)
             (make-point (+ (-> self x?)
                            (-> p2 x?))
                         (+ (-> self y?)
                            (-> p2 y?)))])))
```

> 如果使用的语言支持抽象方法的话，`point`中这些选择器(accessor)方法可以定义为抽象(abstract)的。Smalltalk就可以这么做，这种方法被调用的话就会抛出异常。

请注意，作为一个独立的对象，`point`没有意义，因为它给自己发送自己也不理解的消息。但它可以作为原型，其他点可以扩展之。比如用`make-point`创建的普通点，包含字段`x`和`y`：

```
(define (make-point x-init y-init)
  (OBJECT-DEL point
    ([field x x-init]
     [field y y-init])
    ([method x? () x]
     [method y? () y])))
```

也可以是特殊的点：

```
(define random-point
  (OBJECT-DEL point ()
    ([method x? () (* 10 (random))]
     [method y? () (-> self x?)])))
```

正如我们所说的，这些不同类型的点相互合作，它们都理解`point`原型中定义的消息：

```
> (define p1 (make-point 1 2))
> (define p2 (-> random-point add p1))
> (-> (-> p2 above p1) x?)
8.90016724570533
```

同样，我们可以用委托来共享对象之间的状态。例如，考虑一组共享相同x坐标的点：

```
(define 1D-point
  (OBJECT-DEL point
    ([field x 5])
    ([method x? () x]
     [method x! (nx) (set! x nx)])))

(define (make-point-shared y-init)
  (OBJECT-DEL 1D-point
    ([field y y-init])
    ([method y? () y]
     [method y! (ny) (set! y ny)])))
```

所有由 `make-point-shared` 创建的对象共享同一个父对象 `1D-point` ，由它决定 `x` 坐标。如果改变 `1D-point` ，自然会反映到所有子对象上：

```
> (define p1 (make-point-shared 2))
> (define p2 (make-point-shared 4))
> (-> p1 x?)
5
> (-> p2 x?)
5
> (-> 1D-point x! 10)
> (-> p1 x?)
10
> (-> p2 x?)
10
```

4.4 Self的延迟绑定与模块化

> 参见《Why of Y》。

在 `OBJECT-DEL` 语法抽象的定义中，注意我们在消息发送的定义中使用了自我调用的模式 `(obj obj)` 。我们之前也用到过自我调用模式，是在不赋值的情况下实现递归绑定（译注，参见PLAI）。

> 想想C++和Java等主流语言是怎么做的：它们怎么解决可扩展性(extensibility)和脆弱性(fragility)之间的折衷？

OOP的这个特性也被称为“开放式递归”（open recursion）：任何子对象都可以重新定义其父对象的（父对象的）方法。当然，这种机制有利于可扩展性（extensibility），因为我们可以扩展对象的任何方面，而不必事先预见到需要进行这些扩展。另一方面，开放式递归使得软件变得更加脆弱（fragile），因为以不可预见、不正确的方式扩展对象太过容易。想象一下可能出问题的情况，然后考虑可行的替代设计。为了进一步阐明脆弱性，可以考虑对象的黑盒组合情况：有两个对象，各自独立开发，然后把它们放入委托关系中。可能会出什么问题？

4.5 词法作用域和委托

正如之前所讨论的，在我们的系统中可以定义嵌套的对象。词法嵌套与委托之间的关系蛮有意思的，值得讨论一下。考虑下面的例子：

```
(define parent
 (OBJECT-DEL root ()
   ([method foo () 1])))

(define outer
 (OBJECT-DEL root
    ([field foo (λ () 2)])
    ([method foo () 3]
     [method get ()
             (OBJECT-DEL parent ()
                ([method get-foo1 () (foo)]
                 [method get-foo2 () (-> self foo)]))])))

(define inner (-> outer get))

> (-> inner get-foo1)
2
> (-> inner get-foo2)
1
```

可以看到，自由标识符在词法环境中查找（见 `get-foo1` ），未知消息在委托链上进行查找（见 `get-foo2` ）。这点需要澄清，因为Java程序员习惯的是 `this.foo()` 等同于 `foo()` 。在许多同时支持词法嵌套和某种形式的委托（如继承）的语言中，情况并非如此。

> 其他语言对此有不同的处理。参见Newspeak和AmbientTalk。

Java是怎么处理的？ 试试就知道了！继承链屏蔽（shadow）了词法链：使用 `foo()` 时，如果能在超类中找到方法，则会调用该方法；只有在找不到方法时，才使用词法环境（即 `outer` 对象中的 `foo` ）。因此，对 `outer` 对象的引用是非常脆弱的。这就是为什么Java支持额外的语法形式 `Outer.this` 来引用外层对象。当然，如果直接外层对象的类中找不到方法，那么就继续在它的超类中查找，而不是往词法链上。

4.6 委托模型

我们在这里实现的委托模型只是基于原型的语言的设计空间中的一个点。请自行研究Self，JavaScript和AmbientTalk的文档以了解其设计。你还可以修改我们的对象系统，让其支持不同的模型，比如说JavaScript模型。

4.7 克隆

在我们的语言中（在JavaScript中也是一样），对象都是无中生有的创建的：要么从头创建对象，要么我们有个函数，它的作用是为我们执行对象的创建。历史上，基于原型的语言（如Self）提供了另一种创建对象的方法：克隆(clone)现有对象。

这种方法类似于我们经常对文本（包括代码！）进行的复制—粘贴—修改操作：从某个类似的对象开始，克隆之，然后修改该克隆（比如说，添加方法，更改字段）。

当克隆对象和委托同时存在时，就会出现克隆操作是深（deep）还是浅（shallow）的问题。浅克隆返回的对象和原始对象共享父对象。深克隆返回的对象的父对象是原始对象的父对象的克隆，并依此类推：整个委托链都被克隆。

这里我们不在详细地研究克隆。然而，你应该思考一下，在我们的语言中支持克隆难易如何。由于对象实际上（通过宏展开）被编译成函数，所以问题归结为闭包的克隆。不幸的是，Scheme不支持此操作。出现了源语言和目标语言之间不匹配的情况（想想PLAI第12章）。甘瓜苦蒂！

5 类

回头讨论工厂函数（参见构造对象）：

```
(define (make-point init-x)
  (OBJECT
   ([field x init-x])
   ([method x? () x]
    [method x! (new-x) (begin (set! x new-x) self)])))

(define p1 (make-point 0))
(define p2 (make-point 1))
```

所有点对象都拥有自己的方法，尽管它们是相同的。至少它们的签名和主体是一样的，对吧？它们完全一样吗？事实并非如此。在这个版本的对象系统中，唯一的区别是，方法中包含对象自身：就是说，在 `p1` 的方法中， `self` 指向 `p1` ，而在 `p2` 的方法中它指向 `p2` 。换句话说，方法，也就是函数，因所捕捉的词法环境而不同。

5.1 共享方法定义

为了支持不同的 `self` ，就重复所有的方法定义并不合理。将共同部分（方法体）分解出来，将变量（绑定到 `self` 的对象）参数化更合理。

先试试不用宏来实现。回想一下，不使用宏的情况下，点对象的定义如下：

```
(define make-point
  (λ (init-x)
    (letrec ([self
              (let ([x init-x])
                (let ([methods (list (cons 'x? (λ () x))
                                     (cons 'x! (λ (nx)
                                                 (set! x nx)
                                                 self)))])
                  (λ (msg . args)
                    (apply (cdr (assoc msg methods)) args))))])
      self)))
```

如果将 `(let ([methods...]))` 从 `(λ (init-x) ...)` 中提取出来，我们就可以实现想要的方法定义的共享。但是这样的话，字段变量就不在方法体的作用域中了。具体地说，在这个例子中，这意味着 `x` 在两个方法中都没有绑定。这表明，除了 `self` 之外，方法还需要参数化于状态（字段值）之上。不过，好

在 `self` 可以“持有”状态（它可以捕获其词法环境中的字段绑定）。只要找到通过 `self` 能够提取字段值（还有对其赋值）的方法就可以了。为此，我们的对象将支持两个特定的消息 `-read` 和 `-write` :

```
(define make-point
  (let ([methods (list (cons 'x? (λ (self)
                                   (λ () (self '-read))))
                       (cons 'x! (λ (self)
                                   (λ (nx)
                                     (self '-write nx)
                                     self))))])
    (λ (init-x)
      (letrec ([self
                (let ([x init-x])
                  (λ (msg . args)
                    (case msg
                      [(-read) x]
                      [(-write) (set! x (first args))]
                      [else
                       (apply ((cdr (assoc msg methods)) self) a
rgs)])))])
        self))))
```

请仔细研究这里的方法现在是如何参数化于 `self` 的，还有，要存取字段现在需要向 `self` 发送特殊消息。接下来再研究对象本身的定义：当收到消息时，它首先检查消息是否为 `-read` 或 `-write` ，如果是的话就进行存取操作。来试试这是否可行：

```
(define p1 (make-point 1))
(define p2 (make-point 2))


> ((p1 'x! 10) 'x?)
10
> (p2 'x?)
2
```

5.2 访问字段

当然，这个定义不怎么通用，因为它只适用于一个字段 `x` 。我们需要将其一般化：字段名必须作为参数传给 `-read` 和 `-write` 消息。问题是，如何用字段名（符号）访问对象的词法环境中的同名变量。一个简单的解决方案是使用某种结构来保存字段值。方法的定义就是这样处理的，保存的是方法名称和方法定义之间的关联。不过，与方法表不同，字段绑定是（至少是潜在）可变的。Racket不支持对关联表进行赋值，所以我们使用字典（更确切地说，哈希表），用 `dict-ref` 和 `dict-set!` 访问。

```
(define make-point
  (let ([methods (list (cons 'x? (λ (self)
                                   (λ () (self '-read 'x))))
                       (cons 'x! (λ (self)
                                 (λ (nx)
                                   (self '-write 'x nx)
                                   self))))])
    (λ (init-x)
      (letrec ([self
                 (let ([fields (make-hash (list (cons 'x init-x))
)])
                   (λ (msg . args)
                     (case msg
                       [(-read)  (dict-ref  fields (first args))]
                       [(-write) (dict-set! fields (first args)
                                                   (second args))
]
                       [else
                        (apply ((cdr (assoc msg methods)) self) a
rgs)])))])
        self))))

> (let ((p1 (make-point 1))
        (p2 (make-point 2)))
    (+ ((p1 'x! 10) 'x?)
       (p2 'x?)))
12
```

请注意 `make-point` 现在保存了方法定义的列表，还有，被创建的对象捕获了 `fields` (字段)字典（该字典先初始化，然后返回给对象）。

5.3 类

虽然我们的确实现了方法定义的共享，但是这个解决方案并不理想。为什么？观察对象的定义（上述 `(λ (msg . args) ....)` 的函数体）。在那里实现的逻辑在所有用 `make-point` 创建对象中都是重复的：每个对象都有它自己的副本，当它收到 `-read` 消息时，在 `fields` 字典中查找； `-write` 消息时，更新 `fields` 字典；任何其他消息，查找 `methods` 表，然后应用对应方法。

所以说，所有这些逻辑在对象之间都可以共享。对象体中唯一的自由变量是 `fields` 和 `self` 。换句话说，我们可以把对象定义为它自己外加它的字段，而把所有其他的逻辑都交给 `make-point` 函数。这样的话， `make-point` 的功能不再是单一的只负责创建新的对象，还负责处理对字段的访问和对消息的处理。也就是说， `make-point` 演变成所谓的类（class）。

我们如何表示类？目前它只是可以调用的函数（它会创建对象——一个实例）；如果需要该函数有不同的行为，我们可以应用本书开始时看到的对象模式。

> 在某些语言中，类本身就是对象。这方面的范例就是Smalltalk。绝对值得花时间一学！

于是：

```
(define Point
  ....
  (λ (msg . args)
    (case msg
      [(create) create instance]
      [(read) read field]
      [(write) write field]
      [(invoke) invoke method])))
```

这种模式明确了类的作用：它产生对象，调用方法，读取和写入其实例的字段。

现在，对象的作用是什么？他只需要有标识（identity）功能，知道自己属于哪个类，并记录自己的字段值。它不再自带任何行为。换种说法，对象可以定义为普通的数据结构：

```
(define-struct obj (class values))
```

接下来看看现在该怎么定义Point类：

```
(define Point
  (let ([methods ....])
    (letrec
        ([class
             (λ (msg . vals)
               (case msg
                 [(create) (let ((values (make-hash '((x . 0))))
)
                             (make-obj class values))]
                 [(read) (dict-ref (obj-values (first vals))
                                   (second vals))]
                 [(write) (dict-set! (obj-values (first vals))
                                     (second vals)
                                     (third vals))]
                 [(invoke)
                   (let ((found (assoc (second vals) methods)))
                     (if found
                         (apply ((cdr found) (first vals)) (cddr
 vals))
                         (error "message not understood")))]))])
      class)))

> (Point 'create)
#<obj>
```

要实例化 `Point` 类，只需向其发送 `create` 消息。现在对象是结构体了，我们需要一种方法来发送消息，还有访问其字段。要向对象 `p` 发送消息，先要检索它的类，然后给这个类发送 `invoke` 消息：

```
((obj-class p) 'invoke p 'x?)
```

访问字段也是类似。

5.4 在Scheme中嵌入类

本节我们使用宏在Scheme中嵌入类。

5.4.1 类的宏

我们来定义 `CLASS` 语法抽象，它负责创建类：

```
(defmac (CLASS ([field f init] ...)
               ([method m params body] ...))
     #:keywords  field method
     #:captures self
     (let ([methods (list (cons 'm (λ (self)
                                     (λ params body))) ...)])
       (letrec
           ([class
                (λ (msg . vals)
                  (case msg
                    [(create)
                     (make-obj class
                               (make-hash (list (cons 'f init) .
..)))]
                    [(read)
                     (dict-ref (obj-values (first vals)) (second
 vals))]
                    [(write)
                     (dict-set! (obj-values (first vals)) (secon
d vals) (third vals))]
                    [(invoke)
                     (if (assoc (second vals) methods)
                         (apply ((cdr (assoc (second vals) metho
ds)) (first vals)) (cddr vals))
                         (error "message not understood"))]))])
         class)))
```

5.4.2 辅助语法

我们需要引入新的语法定义，以方便地调用方法（`->`），还需要引入类似的语法，来访问当前对象的字段（`?` 和 `!`）。

```
(defmac (-> o m arg ...)
  (let ((obj o))
    ((obj-class obj) 'invoke obj 'm arg ...)))

(defmac (? fd) #:captures self
  ((obj-class self) 'read self 'fd))

(defmac (! fd v) #:captures self
  ((obj-class self) 'write self 'fd v))
```

还可以定义辅助函数来创建新的实例：

```
(define (new c)
  (c 'create))
```

这个简单的函数在概念上非常重要：它有助于隐藏类在内部作为函数实现的事实，还隐藏了用于请求类创建实例的符号。

5.4.3 例子

来看类的例子：

```
(define Point
 (CLASS ([field x 0])
        ([method x? () (? x)]
         [method x! (new-x) (! x new-x)]
         [method move (n) (-> self x! (+ (-> self x?) n))])))

(define p1 (new Point))
(define p2 (new Point))


> (-> p1 move 10)
> (-> p1 x?)
10
> (-> p2 x?)
0
```

5.4.4 强封装

关于字段访问，我们做了个重要的设计决定：字段访问器 `?` 和 `!` 只能作用于 `self`！即，在我们的语言中不可能访问另一个对象的字段。这被称为具有强封装（Strong Encapsulation）对象的语言。Smalltalk就是这样（访问另一个对象的

字段实际上是发送消息，因此可以由接收方对象来控制）。Java不是：可以访问任何对象的字段（如果可见性(visibility)允许的话）。我们的语法根本不允许访问外部字段。

这样设计的另一个结果是，字段访问只能出现在方法体内：因为接收对象总是 `self` ，所以 `self` 必须已定义。比如说，试试在对象之外用 `?` 读取字段：

```
> (? f)
self: undefined;
 cannot reference undefined identifier
```

更好的做法是，上述程序会产生错误，表明 `?` 未定义。要做到这一点，我们简单地将 `?` 和 `?` 定义为局部语法形式，只在方法体的内被定义，而不是全局范围内有定义。只要将这些字段访问形式的定义从全局移动到 `local` 作用域内， `local` 放在方法定义内：

```
(defmac (CLASS ([field f init] ...)
               ([method m params body] ...))
     #:keywords field method
     #:captures self ? !
     (let ([methods
            (local [(defmac (? fd) #:captures self
                      ((obj-class self) 'read self 'fd))
                    (defmac (! fd v) #:captures self
                      ((obj-class self) 'write self 'fd v))]
              (list (cons 'm (λ (self)
                               (λ params body))) ...))])
                (letrec
                   ([class (λ (msg . vals) ....)]))))
```

在方法列表定义的局部作用域内定义语法形式 `?` 和 `!` ，确保了它们可以在方法体内可用，但在其他地方不可用。

现在，字段访问器方法之外没有定义：

```
> (? f)
?: undefined;
 cannot reference undefined identifier
```

后文统一使用这种局部的方法。

5.5 初始化

我们已经看到，要从类获取对象（即实例化对象）的方法是向类发送 `create` 消息。能够给 `create` 传递参数，以指定对象的字段的初始值通常是有用的。目前，我们的类系统仅支持在类声明时指定默认字段值。在实例化时没法传递初始字段值。

> 初始化方法是Smalltalk编程中的习惯叫法。在Java中，它们被称为构造函数（这可以说是个糟糕的名字，因为我们可以看到，它们并不负责构建对象，只是在实际创建对象之后才对其进行初始化）。

有几种方法可以做到这一点。一个简单的方法是，要求对象实现初始化方法，并让这个类在每个新创建的对象上调用此初始化方法。我们将采用如下约定：如果 `create` 消息没有参数，那么我们不调用初始器（因此使用默认值）。如果有参数传入，我们就用这些参数调用初始器（称之为 `initialize` ）：

```
....
(λ (msg . vals)
  (case msg
    [(create)
     (if (null? vals)
         (make-obj class
                   (make-hash (list (cons 'f init) ...)))
         (let ((object (make-obj class (make-hash))))
           (apply ((cdr (assoc 'initialize methods)) object) val
s)
           object))]
    ....)) ....
```

我们可以改进实例化类的辅助函数，使其接受可变数目的参数：

```
(define (new class . init-vals)
    (apply class 'create init-vals))
```

来试试看：

```
(define Point
 (CLASS ([field x 0])
        ([method initialize (nx) (-> self x! nx)]
         [method x? () (? x)]
         [method x! (nx) (! x nx)]
         [method move (n) (-> self x! (+ (-> self x?) n))])))

(define p (new Point 5))

> (-> p move 10)
> (-> p x?)
15
```

5.6 匿名类，局部类和嵌套类

我们扩展了Scheme，引入了类。扩展的方式类似于之前的对象系统，类表示为一等（first-class）函数。这意味着，我们语言中的类是一等的实体，例如可以作为参数传递（参见前面 `create` 函数的定义）。另外，我们的系统也支持匿名类和嵌套的类。当然，这一切都建立在遵从词法作用域规则的基础上。

```
(define (cst-class-factory cst)
 (CLASS () ([method add (n) (+ n cst)]
            [method sub (n) (- n cst)]
            [method mul (n) (* n cst)])))

(define Ops10  (cst-class-factory 10))
(define Ops100 (cst-class-factory 100))

> (-> (new Ops10) add 10)
20
> (-> (new Ops100) mul 2)
200
```

我们也可以在局部作用域中引入类。也就是说，不同于类是全局可见的一阶实体的语言，我们可以在局部作用域中定义类。

```
(define doubleton
  (let ([the-class (CLASS ([field x 0])
                         ([method initialize (x) (-> self x! x)
]
                          [method x? () (? x)]
                          [method x! (new-x) (! x new-x)]))])
    (let ([obj1 (new the-class 1)]
          [obj2 (new the-class 2)])
      (cons obj1 obj2))))

> (-> (cdr doubleton) x?)
2
```

在这里，引入 `the-class` 的的目的仅在于创建两个实例，然后以对的形式返回这两个实例。在那之后，这个类就不再可用了。换种说法，无法再创建这个类的更多实例了。不过，我们创建的这两个实例当然仍然指向它们的类，因此这些对象仍可以使用。有趣的是，一旦这些对象被垃圾收集，他们的类也可以被收回。

6 继承

既然有了类，我们可能需要一个类似于委托的机制，以便能够重用和选择性地改善现有的类。因此，我们扩展对象系统，支持类继承（class inheritance）。我们将会看到，有许多问题需要处理。像往常一样，我们将逐步讨论。

6.1 类的层次结构

先来引入一个类扩展另一个类的能力（称为它的超类(superclass)）。这里只讨论单一继承（single inheritance），一个类只扩展一个类。

> 多重继承。C++

结果就是，类被组织成层次结构。一个类的所有（传递性的）超类被称为其祖先；对等的，一个类的传递子类（subclass）集称为它的后代。

例如：

```
(define Point
  (CLASS extends Root
           ([field x 0])
           ([method x? () (? x)]
            [method x! (new-x) (! x new-x)]
            [method move (n) (-> self x! (+ (-> self x?) n))])))

(define ColorPoint
  (CLASS extends Point
           ([field color 'black])
           ([method color? () (? color)]
            [method color! (clr) (! color clr)])))
```

6.2 方法查找

当给对象发送消息时，我们在它的类中查找实现此消息的方法，然后调用之。反映到 `CLASS` 宏的定义中就是：

```
[(invoke)
 (if (assoc (second vals) methods)
     (apply ((cdr (assoc (second vals) methods)) (first vals)) (
cddr vals))
     (error "message not understood"))]
```

有了继承，在对象收到一个在其类中找不到方法的消息时，我们可以在超类中寻找方法，并依此类推。首先， `invoke` 协议需要修改，将其分成两步：第一步是 `lookup` （查找），包括当前类中没有找到方法时在超类中进行查找，第二步是实际的 `invoke` 步骤。

```
(defmac (CLASS extends superclass
               ([field f init] ...)
               ([method m params body] ...))
  #:keywords field method extends
  #:captures self ? !
  (let ([scls superclass]
        (methods
         (local [(defmac (? fd) #:captures self
                   ((obj-class self) 'read self 'fd))
                 (defmac (! fd v) #:captures self
                   ((obj-class self) 'write self 'fd v))]
           (list (cons 'm (λ (self)
                            (λ params body))) ...))))
    (letrec ([class (λ (msg . vals)
                      (case msg
                        ....
                        [(invoke)
                         (let ((method (class 'lookup (second va
ls))))
                           (apply (method (first vals)) (cddr va
ls)))]
                        [(lookup)
                         (let ([found (assoc (first vals) method
s)])
                           (if found
                               (cdr found)
                               (scls 'lookup (first vals))))]))]
)
      class)))
```

`CLASS` 语法抽象扩展了，加了 `extends` 子句（这是类定义中新的关键字）。试用这个抽象之前，我们需要在树的顶部定义一个根类，以终结方法查找的过程。如下的 `Root` 类就可以：

```
(define Root
  (λ (msg . vals)
    (case msg
      [(lookup) (error "message not understood:" (first vals))]
      [else     (error "root class: should not happen: " msg)]))
)
```

`Root` 直接实现为函数而不使用 `CLASS` 形式，所以我们无需指定它的超类（它也没有）。如果收到 `lookup` 消息，它会给出消息无法理解的错误。请注意，在此系统中，除了 `lookup` 以外的任何消息发送到根类都是错误。

来看一个非常简单的类继承的例子：

```
(define A
  (CLASS extends Root ()
         ([method foo () "foo"]
          [method bar () "bar"])))
(define B
  (CLASS extends A ()
         ([method bar () "B bar"])))

> (define b (new B))
> (-> b foo)
"foo"
> (-> b bar)
"B bar"
```

看起来都对了：向 `B` 发送其不理解的消息效果正如预期，并且发送 `bar` 的结果是 `B` 中调整过而不是 `A` 中的方法被执行。换一种说法，方法调用被正确的延迟绑定（late binding）。我们说，`B` 中的 `bar` 方法覆盖（override）了 `A` 中定义的同名方法。

再来看个稍微复杂一点的例子：

```
> (define p (new Point))
> (-> p move 10)
> (-> p x?)
10
```

来试试 `ColorPoint` ：

```
> (define cp (new ColorPoint))
> (-> cp color! 'red)
> (-> cp color?)
'red
> (-> cp move 5)
hash-ref: no value found for key
  key: 'x
```

发生了什么？看来，我们不能使用 `ColorPoint` 的 `x` 字段。好吧，我们还没有讨论过在继承中如何处理字段。

6.3 字段和继承

来看一下我们目前是怎么处理对象创建的：

```
[(create)
 (make-obj class
          (make-hash (list (cons 'f init) ...)))]
```

问题就在这里：在字典中我们只初始化了当前类声明的字段的值！还需要对祖先类的字段值进行初始化。

6.3.1 继承字段

对象应该包含其祖先声明的所有字段的值。因此，当创建类时，我们应该确定它的实例的所有字段。要做到这一点，我们必须扩展类，使其保留所有字段的列表，并能够将该信息提供给任何需要的子类。

```
(defmac (CLASS extends superclass
               ([field f init] ...)
               ([method m params body] ...))
  #:keywords field method extends
  #:captures self ? !
  (let* ([scls superclass]
         [methods ....]
         [fields (append (scls 'all-fields)
                         (list (cons 'f init) ...))])
    (letrec
        ([class (λ (msg . vals)
                  (case msg
                    [(all-fields) fields]
                    [(create) (make-obj class
                                        (make-hash fields))]
                    ....))]))))
```

在类的词法环境中，我们引入新的 `fields` 标识符。该标识符绑定到类的实例应该有的全部字段的列表。要获取超类的所有字段，只要向其发送 `all-fields` 消息（其实现简单地返回绑定到 `fields` 的表）。创建对象时，我们就要用这些字段来创建新的字典。

因为我们给类的词汇表增加了新消息，所以需要想想如果 `Root` 收到这个消息该怎么处理：它的所有字段是什么？必须是空表，因为我们不加分辨地使用了 `append` ：

```
(define Root
  (λ (msg . vals)
    (case msg
      [(lookup)     (error "message not understood:" (first vals
))]
      [(all-fields) '()]
      [else (error "root class: should not happen: " msg)])))
```

来试试这是否有效：

```
> (define cp (new ColorPoint))
> (-> cp color! 'red)
> (-> cp color?)
'red
> (-> cp move 5)
> (-> cp x?)
5
```

太好了！

6.3.2 字段的绑定

实际上，还有一个问题我们没有考虑过：如果子类定义了一个字段，其名字已经存在于其祖先之一，会发生什么？

```
(define A
 (CLASS extends Root
       ([field x 1]
        [field y 0])
       ([method ax () (? x)])))
(define B
  (CLASS extends A
        ([field x 2])
        ([method bx () (? x)])))

> (define b (new B))
> (-> b ax)
2
> (-> b bx)
2
```

在这两种情况下，返回的都是绑定到 `B` 的 `x` 字段的值。换句话说，和方法一样，字段也是延迟绑定的。这合理吗？

> 强封装

我们来想一想：对象的目的是将一些（可能可变的）状态封装在适当的程序接口（方法）之后。显然，对方法延迟绑定是理想的，因为方法是对象的外部接口。那么字段呢？字段应该是隐藏的、对象的内部状态——换种说法，实现的细节，而不是公开的接口。其实，请注意我们的语言到目前为止，甚至不能访问另一个对象除 `self` 之外的的字段！那么，至少，对字段的延迟绑定是值得疑问的。

> 私有方法应该延时绑定吗？他们是延迟绑定的吗？

来看一下委托是怎么处理字段的？那里，字段只是函数的自由变量，所以它们遵从词法作用域。对字段来说，这是更合理的语义。在类中定义方法时，其根据该类中直接定义的字段或其超类中的字段。这里的道理是，因为所有这些都是在编写类定义的时候已知的信息。延迟绑定字段意味着对方法中的所有自由变量重新引入了动态作用域：有趣的错误之源和头痛的来源！（想想这样的例子，子类意外地引入与超类中已有名称一样的字段，从而导致混乱。）

6.3.3 字段遮蔽

本节讨论如何定义被称为字段遮蔽（field shadowing）的语义：类的字段遮蔽超类的同名字段，但是方法总是访问它所在的类或其祖先声明的字段。

具体来说，这意味着一个对象可以为同名字段保存不同的值；使用哪一个取决于具体执行的方法在哪个类定义（这被称为方法的宿主类(host class)）。由于这种多重性，只用一个哈希表是不够了。替代方案，我们在类中保存一份字段名称的列表，并在对象中保存由值组成的向量（vector），通过位置访问向量中的值。字段访问将分两步完成：首先根据名称列表确定字段的位置，然后访问对象中值向量对应位置的值。

例如，对于上面的类 `A` ，名称列表是 `'(x y)` ， `A` 一个实例的值向量是 `#(1 0)` 。对于 `B` 类，名称列表是 `'(x y x)` ，一个实例的值向量是 `#(1 0 1)` 。以这种方式保持字段的优点是，在没有遮蔽的情况下，字段总是在对象内相同的位置中。

要遵从遮蔽的语义，我们（至少）有两个选项。一种方法，我们可以将被遮蔽字段重命，例如 `B` 中的字段名变成 `'(x0 y x)` ，这样 `B` 中的方法及其后代只能看到 `x` ——也就是 `B` 中引入的字段——的最新定义。另一种方法是保持字段名不变，查找从字段列表尾部开始：也就是说，我们希望在名称列表中找到字段名最后的位置。这里我们选择后一种方案。

修改 `CLASS` 的定义，以引入向量和字段查找策略：

```
....
[(create)
 (let ([values (list->vector (map cdr fields))])
   (make-obj class values))]
[(read)
 (vector-ref (obj-values (first vals))
             (find-last (second vals) fields))]
[(write)
 (vector-set! (obj-values (first vals))
              (find-last (second vals) fields)
              (third vals))]
....
```

创建对象时，我们用初始字段值构造向量。然后，访问字段时，我们用 `find-last` 返回的位置来访问此向量。不过，试一下就知道，此路不通！语义和之前一样，还是错误的。

为什么呢？回忆一下我们是怎么处理字段访问的，即怎么去除 `?` 语法糖：

```
(defmac (? fd) #:captures self
  ((obj-class self) 'read self 'fd))
```

这里写的表达式是，先询问 `self` 是哪个类，然后发送给该类 `read` 消息。嗯，但是 `self` 是动态绑定到接收方对象的，所以我们总是在要求原来的类访问字段！错误在这里。不应将 `read` 消息发送给接收方的类，而是发送给方法的宿主类。怎么实现呢？需要一种方法，从方法体找到它的宿主类，或者更好的办法，直接访问宿主类的字段列表。

我们可以将字段列表放在方法的词法环境中，就像 `self` 那样，但这样的话程序员可能会意外地影响绑定（与之相反， `self` 一般是面向对象语言中的关键字）。字段列表（以及绑定它的名称）应该是我们的实现内部的东西。既然我们在类中局部定义了 `?` 和 `!` ，可以简单地将字段列表 `fields` 限定在这些语法定义的范围内；由宏观的卫生扩展来确保用户代码不可能意外地影响 `fields` 。

```
....
(let* ([scls superclass]
       [fields (append (scls 'all-fields)
                       (list (cons 'fd val) ...))]
       [methods
        (local [(defmac (? fd) #:captures self
                  (vector-ref (obj-values self)
                              (find-last 'fd fields)))
                (defmac (! fd v) #:captures self
                  (vector-set! (obj-values self)
                               (find-last 'fd fields)
                               v))]
          ....)]))
```

> 这个实现并不理想，因为每次字段访问都会调用 `find-last` （昂贵/线性开销）。可以避免吗？ 如何避免？

请注意，我们现在直接访问 `fields` 表，所以无需再向类发送字段访问消息。对于写入字段也是一样。

来试试这一切是否能按预期运行：

```
(define A
 (CLASS extends Root
       ([field x 1]
        [field y 0])
       ([method ax () (? x)])))
(define B
  (CLASS extends A
        ([field x 2])
        ([method bx () (? x)])))

> (define b (new B))
> (-> b ax)
1
> (-> b bx)
2
```

6.4 清理类协议

我们引入类之后，又对它的协议（protocol）做了不少改变：

- 通过引入 `lookup` 将 `invoke` 协议分成两部分， `lookup` 专门用于在类的层次结构中查找方法定义。
- 为了能够检索类的字段，添加了 `all-fields` 。构建类的时候通过它获取超类的字段列表，追加到当前定义的类的字段列表。
- 去除了字段访问的 `read` / `write` 协议，以便正确地确定方法中的字段名称的作用域。

现在是时候反思一下类协议，看看这里的协议是不是最小化的，还是可以去掉一些部分。判断的标准是什么？既然我们正在讨论类的协议，它最好确实是依赖于类来处理消息。例如，之前介绍的 `read` / `write` 协议就可以删除。回忆一下：

```
....
[(read)  (dict-ref (obj-values (first vals)) (second vals))]
[(write) (dict-set! (obj-values (first vals)) (second vals)
                    (third vals))]
....
```

这里有任何东西依赖于类函数中的自由变量（或者说，依赖于类对象的状态）吗？没有，唯一需要的输入是当前对象、要访问的字段的名称，以及可能写入的值。因此，我们可以直接把这些代码放在 `?` 和 `!` 的展开中，从而有效地“编译掉”一层不必要的解释。

那么 `invoke` 呢？ 来看看，它唯一做的是给自己发送一条消息，这个可以直接在扩展 `->` 时做，这样调用本质上就独立于类了：

```
(defmac (-> o m arg ...)
  (let ([obj o])
    ((((obj-class obj) 'lookup 'm) obj) arg ...)))
```

类协议的其他部分呢？ `all-fields` 、 `create` 和 `lookup` 都访问了类的内部状态： `all-fields` 访问了 `fields` ； `create` 访问了 `fields` 和 `class` 本身； `lookup` 访问了 `methods` 和 `superclass` 。所以，我们的类只需要了解这三种信息。

6.5 发消息给超类

当某个方法覆盖（override）超类中的方法时，有时候需要能调用超类中的定义。允许这么做就可以支持许多典型的改进模式，例如在执行方法之前或之后添加要做的事情，比如对其参数和返回值的进一步处理等等。这被称作给超类发送（super send）。我们选择 `-->` 作为给超类发送的语法。

先来看一个例子：

```
(define Point
 (CLASS extends Root
         ([field x 0])
         ([method x? () (? x)]
          [method x! (new-x) (! x new-x)]
          [method as-string ()
                  (string-append "Point("
                                 (number->string (? x)) ")")]))
)

(define ColorPoint
 (CLASS extends Point
         ([field color 'black])
         ([method color? () (? color)]
          [method color! (clr) (! color clr)]
          [method as-string ()
                  (string-append (--> as-string) "-"
                                 (symbol->string (? color)))]))
)

> (define cp (new ColorPoint))
> (-> cp as-string)
"Point(0)-black"
```

请注意，给超类发送使我们能够在 `ColorPoint` 的定义中重用和扩展 `Point` 中 `as-string` 的定义。在 `Java` 中，这是通过对 `super` 调用方法来完成的，但究竟 `super` 是什么？给超类发送的语义是什么？

首先要澄清的是：给超类发送的接收者是啥？在上面的例子中，当使用 `-->` 时， `as-string` 发送给了哪个对象？ `self` ！事实上， `super` 只影响了方法查找。一个常见的误解是，在执行给超类发送时，方法查找从接收方的超类开始，而不是从它的类开始。我们来构造一个小例子，看看为什么这是不正确的：

```
(define A
  (CLASS extends Root ()
        ([method m () "A"])))

(define B
  (CLASS extends A ()
        ([method m () (string-append "B" (--> m) "B")])))

(define C
  (CLASS extends B () ()))

(define c (new C))
(-> c m)
```

这个程序返回什么？我们来研究一下。 `->` 展开为发送 `lookup` 给 `c` 的类，也就是 `C` 。在 `C` 中没有 `m` 方法，所以转而发送 `lookup` 给其超类， `B` 。 `B` 找到 `m` 对应的方法，并返回之。下一步调用此方法，第一个参数是当前的 `self` （也就是 `c` ），接下来是消息的参数，在这里为空。对这个方法求值就需要对 `string-append` 的三个参数求值，其中第二个参数是给超类发送。如果使用上述给超类发送的定义，那么 `m` 不是在 `C` （接收方的实际类）中查找，而是在 `B` （它的超类）中查找的。 `B` 中有 `m` 方法吗？是的，我们正在执行的就是它……换句话说，如果这么理解 `super` ，上述程序将不会终止。

> 一些动态语言，比如Ruby，允许在运行时改变类的继承关系。这在基于原型的语言（如Self和JavaScript）中很常见。

错在哪里？给 `self` 发送时，不应该在接收方的超类中查找方法。在这个例子中，我们应该在 `A` 而不是在 `B` 中查找 `m` 。为此，我们需要知道执行给超类发送的方法的宿主类的超类。这个值应该是在方法体中静态绑定还是动态绑定的？我们刚才已经说过了：它是方法的宿主类的超类，不可能动态改变（至少在我们的语言中如此）。好在在方法的词法环境中，已经有了指向超类的绑定， `scls` 。所以，我们只需要引入新的局部宏 `-->` ，其展开请求超类 `scls` 来查找消息。 `-->` 可以被用户代码使用，所以它要被添加到 `#:captures` 标识符列表中：

```
(defmac (CLASS extends superclass
               ([field f init] ...)
               ([method m params body] ...))
  #:keywords field method extends
  #:captures self ? ! -->
  (let* ([scls superclass]
         [fields (append (scls 'all-fields)
                         (list (cons 'f init) ...))]
         [methods
          (local [(defmac (? fd) ....)
                  (defmac (! fd v) ....)
                  (defmac (--> md . args) #:captures self
                    (((scls 'lookup 'md) self) . args))]
            ....)]))))
```

请注意， `lookup` 现在被发送到当前正在执行的方法的宿主类的超类 `scls` ，而不是当前对象的实际类。

```
> (define c (new C))
> (-> c m)
"BAB"
```

6.6 继承和初始化

之前已经讨论过，通过引入称为初始器的特殊方法，来初始化对象。一旦对象被创建，在被返回给创建者之前，需要调用它的初始器。

现在有了继承，这个过程变复杂了一点，因为如果初始器能相互覆盖，可能会忽略一些必要的初始化工作。初始器的工作可能非常具体，我们希望避免子类必须处理所有的细节。可以假定其语义和一般方法的语义一样，那么子类中的 `initialize` 可以根据需要调用超类的初始器。这种自由导致的问题是，在继承的字段还没有一致地初始化时，子类中的初始器就可能开始处理对象了。为了避免这个问题，在Java中，构造函数做的第一件事必须是调用超类的构造函数（它可以先计算此调用的参数，仅此而已）。即使不在源代码中明确写出，编译器也会添加这个调用。事实上，在VM（虚拟机）层面字节码验证器也会检验这一点：因此，底层的节码操作也无法绕开对超类构造函数的调用。

7 充满可能的世界

本书在Scheme中简单地逐步构建了对象系统，但我们只阐述了面向对象编程语言的一些基本概念。在语言设计中，总是存在各种各样的可能性有待探索，比如同样的想法的变种、延伸等。

这里给出一些（有限的/随意挑选的）特性和机制，你可以在某些现有的面向对象编程语言中找到，但在我们的讨论中没有涉及。你可以试试将其集成到对象系统中。当然，更有意思的是，你该自己去思考其他特性，还有研究现有的语言并弄清楚如何整合其独特的特性。

- 方法的可见性：public / private
- 声明覆盖超类中方法的方法：override
- 声明不能被覆盖的方法：final
- 声明预期将被继承的方法：inherit
- 可扩展的方法：inner
- 接口（Interface）：能理解的消息的集合
- 检查某个对象是否是某个类的实例的协议，检查某个类是否实现某个接口的协议，……
- 超类的正确初始化协议，实名初始化属性
- 多重继承
- Mixins
- Traits
- 类作为对象，元类（metaclass），……

还有许多优化，例如：

- 计算字段的偏移量（offset），以直接访问字段
- 用于直接方法调用的虚函数表（vtable）和索引（indice）

这里以习题的形式介绍两种机制，接口和mixin，以及它们的组合（即使用接口实现mixin规范）。

7.1 接口

（在我们的语言中）实现新的语法形式，定义接口，接口可以扩展超接口：

```
(interface (superinterface-expr ...) id ...)
```

实现新的语法形式，类可以实现（多个）接口：

```
(CLASS* super-expr (interface-expr ...) decls ...)
;decls为类主体中的申明
```

例如：

```
(define positionable-interface
  (interface () get-pos set-pos move-by))

(define Figure
  (CLASS* Root (positionable-interface)
     ....))
```

扩展类的协议，使之能检查某个类是否实现了某个接口：

```
> (implements? Figure positionable-interface)
#t
```

7.2 Mixin

Mixin是将超类参数化的类声明。当类的继承层次结构中存在共享部分，而单继承又不足以表达时，mixin可以组合创建新类。

因为我们的类由函数实现的，是一等的值（first-class value），所以mixin的实现是“免费的”。

```
(define (foo-mixin cl)
  (CLASS cl (....) (....)))

(define (bar-mixin cl)
  (CLASS cl (....) (....)))

(define Point (CLASS () ....))

(define foobarPoint
  (foo-mixin (bar-mixin Point)))
(define fbp (foobarPoint 'create))
....
```

Mixin和接口结合，可以检查给定的基类是否实现了一组特定的接口。定义MIXIN语法形式：

```
(MIXIN (interface-expr ...) decl ...)
```

这应该是个函数，其输入是基类，先检查该基类实现了所有指定的接口，然后返回（用给定的声明）扩展基类所得的新类。